Counterexample Guided Abstraction Refinement of Product-Line Behavioural Models Technical Report

Maxime Cordy*
PReCISE Research Center
University of Namur, Belgium
mcr@info.fundp.ac.be

Patrick Heymans
PReCISE Research Center,
University of Namur, Belgium.
phe@info.fundp.ac.be

Axel Legay
INRIA Rennes, France
axel.legay@inria.fr

Pierre-Yves Schobbens
PReCISE Research Center,
University of Namur, Belgium.
pys@info.fundp.ac.be

Bruno Dawagne
PReCISE Research Center,
University of Namur, Belgium.
bdawagne@student.fundp.ac.be

Martin Leucker
University of Lübeck, Germany
leucker@isp.uni-luebeck.de

ABSTRACT

The model-checking problem for Software Products Lines (SPLs) is harder than for single systems: variability constitutes a new source of complexity that exacerbates the state-explosion problem. Abstraction techniques have successfully alleviated state explosion in single-system models. However, they need to be adapted to SPLs, to take into account the set of variants that produce a counterexample. In this paper, we apply CEGAR (Counterexample-Guided Abstraction Refinement) and we design new forms of abstraction specifically for SPLs. We carry out experiments to evaluate the efficiency of our new abstractions. The results show that our abstractions, combined with an appropriate refinement strategy, hold the potential to achieve large reductions in verification time, although they sometimes perform worse. We discuss in which cases a given abstraction should be used.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

Keywords

Software Product Lines, Model Checking, CEGAR, Abstraction, Features

1. INTRODUCTION

Variability is ubiquitous in today's systems, be it in the form of configuration options or extensible architectures. By mastering variability, developers can adapt their system to changing requirements without having to develop entirely new applications. Software Product Lines (SPLs) are a popular form of variability-intensive systems. They are families of similar software systems developed together to make economies of scale [19]. SPL engineering aims to facilitate the development of the members of a family (called *products* or *variants*) by identifying upfront their commonalities and differences. Variability in SPLs is commonly represented in terms of *features*, *i.e.*, units of difference between products that appear natural to stakeholders. Each product of an SPL is therefore defined by its set of features. Hierarchies of features and dependencies between features (*e.g.*, requires, excludes) are typically captured in a *Feature Model* (FM), *i.e.* a tree-like structure that specifies which combinations of features are valid [30, 38] (see Figure 3 in Section 2 for an example).

Nowadays, SPL engineering is widespread in industry, including critical areas like automotive and avionics. The emergence and the increasing popularity of SPLs have raised the need for SPL-specific quality assurance techniques. Indeed, engineers have to provide solid evidence that *all* the products they build satisfy their intended requirements. Moreover, in case of failure, they should identify which features, or combinations of features, are responsible for the errors in order to facilitate repair.

Model checking is an automated technique to verify a behavioural model of a system against a property expressed in temporal logic [12, 5]. It relies on an exhaustive exploration of the model in search for counterexamples, i.e., executions that violate the property to verify. Due to its exhaustiveness, model checking is costly in time and memory. When applied to real systems with a typically huge state space, model checking faces a combinatorial blow-up called *state explosion*. The model-checking problem is even harder for SPLs: in this case, the model checker must either prove the absence of errors or find a counterexample for *each* variant that can produce a violation. Given that the worst-case number of products of an SPL is exponential in the number of features, variability dramatically exacerbates state explosion. As a consequence, it is not feasible to apply single-system model checking to the thousands of

*FNRS Research Fellow

FSE 2014, November 16–22, Hong Kong
.

variants that can compose real-world SPLs.

In recent years, many *variability-aware* techniques have been designed to address the SPL model checking problem [28, 18, 17, 2, 4]. These techniques keep track of variability information contained in an SPL behavioural model to associate each execution path to the exact set of variants able to produce it. By doing so, they are able to identify the set of products that violate a given property, and to report a counterexample of violation for each of them. Moreover, being aware of variability allows them to check behaviour common to several products only once. This is a clear improvement over an enumerative application of single-system model checking, which verifies a behaviour as many times as there are products that can exhibit it.

Although earlier experiments suggest that these techniques bring substantial performance gains [16, 3], further improvements are required to verify industrial SPLs. First, SPL model-checking methods still suffer from the state-explosion problem inherent to model checking. Second, their practical time complexity still grows more than linearly with the number of features [14]. In single-system verification, one of the most effective answers to state explosion is *model abstraction*, which creates more concise – therefore easier to verify – models of the system, typically by merging similar states. This reduced size often comes at the cost of inaccuracies in the models, thereby affecting the properties they satisfy. A reported counterexample can therefore be *spurious*, that is, it exists within the abstract model but the not in the real, concrete model. In this case, the abstraction must be refined to eliminate this false positive. Common methods to achieve this refinement make use of the spurious counterexample itself. They give rise to Counterexample Guided Abstraction Refinement (CEGAR), *i.e.* abstraction techniques that iteratively refine an abstract model until either they find a real counterexample or they can prove the absence of violation [11].

In spite of their success in single-system model checking, abstraction techniques for SPLs have received little attention (see more in Section 6). In this paper, we fill this gap and propose SPL-specific abstraction procedures based on CEGAR. Applying CEGAR to SPLs is more tedious because a counterexample can be real for some products and spurious for others. This observation leads us to two refinement strategies: one refines the model as soon as it finds a spurious counterexample, whereas the other performs the spuriousity check and the refinement after the discovery of all the counterexamples. As for the abstraction of the model, we distinguish between (1) *state abstraction* that only merge states as in single-model abstraction, (2) *feature abstraction* that modifies only the variability information contained in the model, and (3) *mixed abstraction* that combines the previous two types. This latter type is the most complicated to implement, as spuriousness can originate from the merging of states, the abstraction of features, or both. Throughout the paper, we systematically prove the correctness of our approach on the basis of mathematical relations such as simulation relations. We implemented both abstractions and their combination in ProVeLines, an SPL model checker we developed [15, 23]. We carried out experiments to evaluate the efficiency of different combinations of refinement strategies and abstractions. Our results tend to show that state abstraction brings performance gains most of the time, whereas feature abstraction generally results in small losses of performance but achieve huge decreases of verification time in some cases. Preliminary experiments on mixed abstraction tend to show that its performance is comparable to that of state abstraction, although slightly worse on average. Other abstractions of this kind could, however, be designed as part of future work and yield better results.

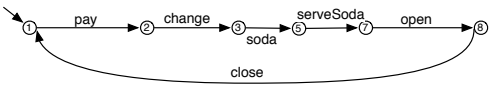


Figure 1: A TS modelling a vending machine

The structure of the paper is as follows. Section 2 recapitulates essential background. We present an overview of our CEGAR procedures and refinement strategies in Section 3. In Section 4, we show how to build the three forms of abstraction functions and to detect spurious counterexample in each case. We describe our implementation and report evaluation results in Section 5. Finally, we discuss related work in Section 6.

2. BACKGROUND

In this section, we recapitulate established concepts related to the verification and abstraction of single-system behavioural models. We also recall some definitions of our previous work that are needed in this paper.

2.1 Counterexample Guided Abstraction Refinement

Model checking is an established technique for verifying both hardware and software against temporal properties [12, 5]. Basically, given the model of a system M and a temporal property Φ, a model-checking algorithm determines whether or not M satisfies Φ, written $M \models \Phi$. For single systems, a transition system (TS) is commonly used as a model for the system. It is defined as follows.

Definition 1 [5] *A TS is a tuple* $(S, Act, trans, I, AP, L)$ *where* S *is a set of states,* Act *is a set of actions,* $trans \subseteq S \times Act \times S$ *is the transition relation,* $I \subseteq S$ *is a set of initial states,* AP *is a set of atomic propositions, and* $L : S \to 2^{AP}$ *is a labelling function that associates every state with the set of atomic propositions satisfied by this state. We assume that every TS has no terminal state, i.e. a state without outgoing transition.*

Figure 1 shows a TS representing a soda vending machine. For clarity, we did not display the atomic propositions satisfied by each state.

In what follows, we also denote $(s, \alpha, s') \in trans$ by $s \xrightarrow{\alpha} s'$. An execution of the model is an alternating infinite sequence of states and actions that satisfies the transition relation, *i.e.* a sequence $s_0\alpha_0 s_1\alpha_1 \ldots$ with $s_i \xrightarrow{\alpha_i} s_{i+1}$ for any $i \geq 0$. A trace (also called *behaviour*) of the system is the sequence of atomic propositions satisfied during one of its executions. The semantics of a TS, noted $[\![ts]\!]_{TS}$, is then its set of behaviours:

$$[\![ts]\!]_{TS} = \{L(s_0), L(s_1), \ldots \mid s_0 \in I \wedge (s_i \xrightarrow{\alpha_i} s_{i+1})\}.$$

After building a TS, one can verify it against a property expressed in temporal logic. In this paper, we particularly focus on *Linear Time Logic* (LTL) [36]. For instance, a desired property for the vending machine is that it

will eventually serve soda each time soda is ordered, noted $\Box(soda\_ordered \Rightarrow \Diamond soda\_served)$ in LTL. Given a TS and a property Φ, a model checker reports either that the property holds in the model or a *counterexample*, *i.e.* an execution of the model that violates the property. In our example, the TS indeed satisfies the aforementioned property since it will necessarily reach state 7 after state 5.

TS can model a software product at different abstraction levels. If a more abstract (that is, smaller) model preserves the properties of a larger model, it is more efficient to check the properties on the abstract model. It is therefore essential to relate two models from different abstraction levels. For single systems, this information is formally captured by a *simulation relation* [33].

Definition 2 [33, 5] *Let* $TS_i = (S_i, trans_i, I_i, AP, L_i), i \in \{1,2\}$ *be two TS over* AP. *A simulation for* (TS_1, TS_2) *is a binary relation* $\mathcal{R} \subseteq S_1 \times S_2$ *such that*

1. $\forall s_1 \in I_1 \bullet \exists s_2 \in I_2 \bullet (s_1, s_2) \in \mathcal{R}$ *and*
2. $\forall (s_1, s_2) \in \mathcal{R}$ *it holds that* $L_1(s_1) = L_2(s_2)$ *and* $\forall s_1' \in Post(s_1) \bullet \exists s_2' \in Post(s_2) \bullet (s_1', s_2') \in \mathcal{R}$.

where $Post(s) = \{s' \mid \exists \alpha \bullet s \xrightarrow{\alpha} s'\}$ *denotes the set of states that can be reached from* s. *Then,* TS_2 *simulates* TS_1, *denoted by* $TS_1 \preccurlyeq_{TS} TS_2$ *iff there exists a simulation for* (TS_1, TS_2).

According to this definition, if TS_2 simulates TS_1, then TS_2 can reproduce any behaviour of TS_1. Simulation can characterise the behaviour of an abstract transition system $\widehat{ts}$ with regard to an original model ts. Informally, an abstract transition system is obtained by merging states for which a so-called abstraction function returns the same value. The abstraction may add or remove behaviour, depending on the chosen abstraction function. However, a relevant analysis requires to have either $ts \preccurlyeq_{TS} \widehat{ts}$, $\widehat{ts} \preccurlyeq_{TS} ts$ or both. Under this condition, the abstraction preserves the (un)satisfiability of properties expressed in particular logics. In particular, an LTL formula satisfied by the simulating TS is preserved in the simulated one.

Property 3 [33, 5] *Let* TS_1 *and* TS_2 *be two transition systems and* Φ *an LTL property. Then,*

$$TS_1 \preccurlyeq_{TS} TS_2 \Rightarrow (TS_2 \models \Phi \Rightarrow TS_1 \models \Phi).$$

In other words, if TS_1 does not satisfy Φ, neither does TS_2. In particular, if TS_2 is an abstraction of TS_1, proving that the abstract TS_2 verifies an LTL formula suffices to ensure that the formula holds for TS_1. Therefore, abstraction can drastically shorten the time and space cost of verification. In this paper, we consider *existential* abstraction functions.

Definition 4 [11] *An existential abstraction* h *is a surjection* $h : S \to \widehat{S}$ *such that* $h(s) = h(s') \Rightarrow L(s) = L(s')$. *An abstraction of a TS under* h *is* $TS_h = (\widehat{S}, Act, trans_h, I_h, AP, L_h)$ *where* $\widehat{S} = \{h(s) | s \in S\}$, $I_h = \{h(s_0) | s_0 \in I\}$, $L_h(h(s)) = L(s)$, *and* $trans_h$ *is defined such that* $s \xrightarrow{\alpha} s' \Rightarrow h(s) \xrightarrow{\alpha} h(s')$.

Existential abstractions can facilitate verification thanks to two interesting properties. First, TS_h is such that $TS \preccurlyeq_{TS} TS_h$ [11]. Second, they can be defined such that $\widehat{S}$ is significantly smaller than S. Therefore, for a given LTL formula Φ we can prove that the concrete TS satisfies Φ by proving that TS_h satisfies Φ. However, if TS_h does not satisfy Φ then TS may still satisfy Φ, that is, the reported counterexample is *spurious*. This may occur when TS_h has strictly more behaviour than TS. In this case, it is required to *refine* the abstraction in order to eliminate the spurious counterexample. From the refinement, we obtain a new abstraction function h' which is less coarse than h, that is, $TS \preccurlyeq_{TS} TS_{h'} \preccurlyeq_{TS} TS_h$.

Counterexample Guided Abstraction Refinement (CEGAR) is an established refinement method that consists in using a spurious counterexample to refine the abstraction [11]. From the concrete TS, we build an initial abstraction based on a given existential abstraction function. We then feed the abstract model into the model checker together with the property to check. If the abstraction satisfies the formula, then so does the concrete TS. Otherwise, we replay the counterexample on the concrete system to determine whether it is spurious. If it is not, then the concrete TS violates the formula. Otherwise, we refine the abstraction on the basis of the counterexample and we repeat the process.

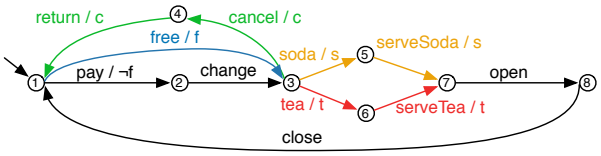


Figure 2: A FTS modelling a vending machine SPL

2.2 Featured Transition Systems

Although TS are suitable to model the behaviour of individual systems, they cannot represent the behaviour of an SPL and link each behaviour to the exact set of products able to execute it. To overcome this, we defined *Featured Transition Systems* (FTS) [18]. Basically, an FTS is a TS where each transition receives an additional label that specifies which combinations of features are required to trigger the transition. These features are declared in an FM that establishes the set of legal feature combinations [30, 38], *i.e.* the *valid* products of the SPL. For this paper, it is enough to know that the semantics of a FM d defined over a set of features F is the set of all the valid products, that is a set of sets of features, denoted by $[\![d]\!]_{FM} \subseteq 2^F$. For a more formal definition of FMs, see Schobbens *et al.* [38]. Formally, FTS are defined as follows.

Definition 5 [17] *An FTS is a tuple* $(S, Act, trans, I, AP, L, d, \gamma)$, *where* $S, Act, trans, I, AP, L$ *are defined as in Definition 1,* d *is a FM over features* F, *and* $\gamma : trans \to \mathbb{B}(F)$ *is a total function labelling each transition with a* feature expression, *i.e. a Boolean function over the set of features. By* $[\![\gamma(t)]\!]$, *we denote the set of products that satisfy* $\gamma(t)$.

Figure 2 shows an FTS modelling a vending machine SPL which includes the vending machine presented in Figure 2. We observe that transition $1 \xrightarrow{free} 3$ is labelled with feature f, meaning that feature f is required to trigger the transition. On the contrary, $1 \xrightarrow{pay} 2$ is labelled with $\neg f$, meaning that feature f must not be present to trigger the transition.

Note that a feature model can also be regarded as a feature expression, i.e., the formula encoding all the constraints it expresses. Henceforth, we use d to denote this feature ex-

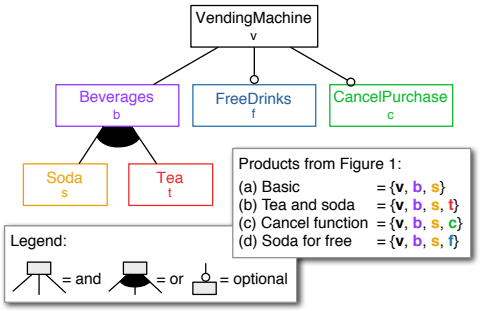


Figure 3: The FM modelling the variability of the vending machine SPL

pression as well. Thanks to the labelling function γ, an FTS encodes the behaviour of a set of products. More precisely, the TS modelling a given product p, noted $FTS_{|p}$, is obtained by removing all the transitions that p cannot execute. This operator is called *projection* [18]. For instance, the TS shown in Figure 1 is the projection of the vending machine FTS onto product $\{v, s\}$. The semantics of an FTS, noted $[\![.]\!]_{FTS}$, is therefore a function that associates each valid product with the semantics of the projection of the FTS onto that product. In this paper, we use FTS to model the behaviour of SPLs; being a fundamental formalism, it can be regarded as a unified semantics for higher-level SPL behavioural modelling languages.

The model checking problem for SPLs is different from its single-system counterpart. Its objective is indeed to identify *all* the product variants that can execute a counterexample. We previously designed efficient algorithms to check an FTS against LTL formulae [18, 16]. During an exploration, these algorithms keep track of the feature expressions met along the transitions. This means that they separate the verification of different sets of products only if they discover a behavioural discrepancy between them. This optimisation is called *late splitting* [3]. Furthermore, the accumulated feature expressions allow to associate a found counterexample to the exact set of products that execute it. An immediate consequence is that the model checker may have to discover *multiple* counterexamples to identify all the violating products. Unlike single-system model checking, the search thus cannot stop after one counterexample is found. This is a fundamental difference that has a real importance in the design of abstraction methods for SPL model checking.

It may happen that a property is relevant only to a subset of the products. Hence, we extended LTL with a *feature quantifier* [18]. A formula of the resulting logic, called fLTL, has the form $[\chi]\Phi$ where χ is a feature expression and Φ is an LTL formula. Intuitively, it means that Φ has to hold only for valid products described by χ. Formally, the set of products satisfying $[\chi]\Phi$ is

$$[\![FTS \models [\chi]\Phi]\!] = \{p \in [\![d]\!]_{FM} \mid p \in [\![\chi]\!] \Rightarrow FTS_{|p} \models \Phi\}.$$

As a simulation relation is a prerequisite to prove the correctness of abstraction-based model checking, we need to extend this concept to make it applicable to FTS and aware of variability. This new relation, named *F-simulation*, is defined as follows.

Definition 6 [21] *Let* $fts_i = (S_i, Act_i, trans_i, I_i, AP, L_i, d, \gamma_i)$, $i \in \{1, 2\}$, *be two FTS. Then, the set of valid products for which* fts_1 *is simulated by* fts_2, *is*

$$[\![fts_1 \preccurlyeq_{FTS} fts_2]\!] = \{p \in [\![d]\!]_{FM} : fts_{1\,|p} \preccurlyeq_{TS} fts_{2\,|p}\}.$$

fts_1 *is simulated by* fts_2 *iff* $[\![fts_1 \preccurlyeq_{FTS} fts_2]\!] = [\![d]\!]_{FM}$.

If fts_2 simulates fts_1, then for each valid product p every behaviour of p in fts_1 is also a behaviour of p in fts_2. Therefore, any fLTL formula satisfied in fts_2 is also satisfied in fts_1. Efficient computation algorithms are found in [22]. For this paper, we just need its definition.

3. CEGAR STRATEGIES IN SPL MODEL CHECKING

The only way to apply existing abstraction techniques [27, 11, 13] to SPLs is to execute them either product by product or on a product simulator [2]. These methods thus cannot address the SPL model-checking problem as we defined it in the previous section. There are two fundamental differences between single-system and SPL model checking. First, a model for *multiple* products, *viz.* an FTS, is checked as opposed to a single-system model, *viz.* a TS. This implies that (1) an appropriate abstraction function must preserve the behaviour of all the products modelled by the FTS, and (2) this function can modify the labelling function γ. To produce FTS abstractions, we can thus either merge states as in single-system abstraction, weaken feature expressions to make transitions available to more products, or both. These three solutions, shown in Figure 4, are respectively called *state abstraction*, *feature abstraction*, and *mixed abstraction*. A second requirement is that the CEGAR process cannot stop after only one real counterexample is found if this counterexample is not executable by all products. If it does stop, the model checker could ignore violations performed by other products; the SPL model checking problem would thus not be answered appropriately. To address these requirements, we first extend the definition of existential abstraction, a common abstraction method that does not remove existing behaviours, and we provide property preservation proofs. Then we present two SPL-specific CEGAR procedures that can verify all the products.

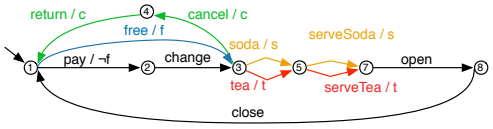
As before, we consider existential abstraction functions as these guarantee behaviour preservation. To transpose this concept to FTS while maintaining the preservation property, we add another requirement to the abstraction function: any product that can execute a concrete transition must be able to execute the corresponding abstract transition. This leads to the following new definition of existential abstraction.

Definition 7 *An F-abstraction is a surjection* $h : S \to \widehat{S}$ *such that* $h(s) = h(s') \Rightarrow L(s) = L(s')$. *An abstraction of a FTS under* h *is* $FTS_h = (\widehat{S}, Act, trans_h, I_h, AP, L_h, d, \gamma_h)$ *where* $\widehat{S}$, Act, $trans_h$, I_h, AP, L_h *are as in Definition 4 and* γ_h *is such that*

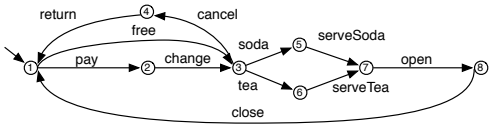
$$\Big(\bigvee_{s \in h^{-1}(\widehat{s}), s' \in h^{-1}(\widehat{s'}) \bullet s \xrightarrow{\alpha} s'} \gamma(s, \alpha, s')\Big) \Rightarrow \gamma_h(\widehat{s}, \alpha, \widehat{s'}) \quad (1)$$

or equivalently

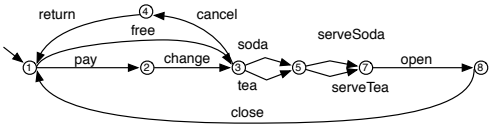
$$\Big(\bigcup_{s \in h^{-1}(\widehat{s}), s' \in h^{-1}(\widehat{s'}) \bullet s \xrightarrow{\alpha} s'} [\![\gamma(s, \alpha, s')]\!]\Big) \subseteq [\![\gamma_h(\widehat{s}, \alpha, \widehat{s'})]\!].$$



(a) State abstraction



(b) Feature abstraction



(c) Mixed abstraction

Figure 4: Examples of the three forms of abstraction.

Intuitively, h is responsible for state abstractions, whereas γ_h creates feature abstractions. Any FTS obtained from an F-abstraction simulates the concrete FTS for all products.

Theorem 8 *Let h be an F-abstraction function. Then we have $[\![FTS \preccurlyeq_{FTS} FTS_h]\!] = [\![d]\!]_{FM}$.*

Proof. *Since existential abstractions from TS preserve the behaviour of the concrete TS modulo simulation (see Section 2), any given transition of FTS is also a transition of FTS_h. It thus remains to prove that this transition is available in FTS_h for at least the same set of products, that is, for any states s, s' we have*

$$(s, \alpha, s') \in trans_{|p} \Rightarrow (\widehat{s}, \alpha, \widehat{s'}) \in (trans_h)_{|p}$$

where $trans_{|p}$ and $(trans_h)_{|p}$ denote the transitions of $FTS_{|p}$ and $(FTS_h)_{|p}$, respectively. This property directly follows from the definition of γ_h:

$$\begin{aligned}
&\Big(\bigvee_{s,s' \bullet s \xrightarrow{\alpha} s'} \gamma(s,\alpha,s')\Big) \Rightarrow \gamma_h(\widehat{s},\alpha,\widehat{s'}) \\
\Leftrightarrow &\bigwedge_{s,s' \bullet s \xrightarrow{\alpha} s'} \gamma(s,\alpha,s') \Rightarrow \gamma_h(\widehat{s},\alpha,\widehat{s'}) \\
\Rightarrow &\bigwedge_{s,s' \bullet s \xrightarrow{\alpha} s'} p \in [\![\gamma(s,\alpha,s')]\!] \Rightarrow p \in [\![\gamma_h(\widehat{s},\alpha,\widehat{s'})]\!] \\
\Leftrightarrow &\forall s, s' \bullet (s,\alpha,s') \in trans_{|p} \Rightarrow (\widehat{s},\alpha,\widehat{s'}) \in (trans_h)_{|p}
\end{aligned}$$

with $s \in h^{-1}(\widehat{s})$ and $s' \in h^{-1}(\widehat{s'})$. Hence $[\![FTS \preccurlyeq_{FTS} FTS_h]\!] = [\![d]\!]_{FM}$. □

We are now ready to present our CEGAR procedures for FTS. Given that an FTS model checker may return several counterexamples, two refinement strategies can be followed. The first strategy, called *Find All Before Refining (FABR)* consists in waiting for the model checker to find all the counterexamples it should return, then checking the spuriousity of each of them, and refining the model if need be. Conversely, the second strategy, *Refine When Found One (RWFO)*, checks spuriousity and possibly refines the model as soon as the model checker finds one counterexample.

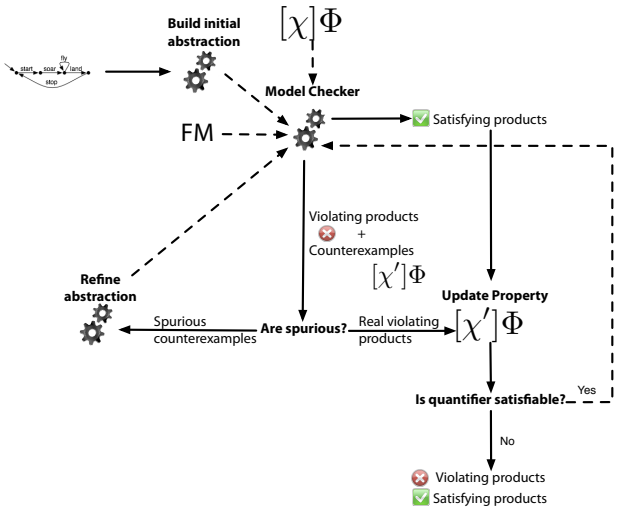


Figure 5: Overview of FABR

Figure 5 gives an overview of FABR. First, we apply an initial F-abstraction (h, γ_h) to obtain an abstract FTS which is given to the model checker along with the FM and an fLTL formula $[\chi]\Phi$. The initial abstraction is built according to a chosen abstraction technique (see Section 4). The model checker returns (1) the set of products satisfying the formula, and (2) the set of products violating the formula together with counterexamples. We then check whether some of these counterexamples are spurious. If that is the case, we refine the abstraction to build a new F-abstraction $(h', \gamma_{h'})$ such that

$$[\![FTS_{h'} \preccurlyeq_{FTS} FTS_h]\!] = [\![d]\!]_{FM} \quad (2a)$$
$$[\![FTS \preccurlyeq_{FTS} FTS_{h'}]\!] = [\![d]\!]_{FM} \quad (2b)$$
$$[\![FTS_h \preccurlyeq_{FTS} FTS_{h'}]\!] \subset [\![d]\!]_{FM} \quad (2c)$$

and we verify $FTS_{h'}$ against the property. During this new search, we can safely ignore the products that were previously recognized as satisfying. Indeed, by definition of F-simulation any product that satisfies $[\chi]\Phi$ in FTS_h also satisfies the formula in $FTS_{h'}$ and FTS. We can also ignore products that can execute a *real* counterexample, since the new check will report them as violating as well. To prevent the model checker to consider these products, it suffices to transform the feature quantifier χ into $\chi' = \chi \wedge \neg\upsilon$ where υ is the feature expression characterizing the products to ignore. We repeat the process until the model checker returns no spurious counterexample, or equivalently while the updated feature quantifier χ' is satisfiable. As a result, we pinpoint the products that satisfy the formula and those that do not.

The RWFO strategy is illustrated in Figure 6. As before, we build an initial abstraction and model-check it. If all the products satisfy the formula, we stop. Otherwise, we stop the verification as soon as the model checker finds a counterexample. If this counterexample is spurious, we refine the abstraction and start the model checking again. Otherwise, we update the formula to ignore the products that can execute the counterexample. If the resulting feature quantifier χ' is satisfiable, we *carry on the verification procedure from when the counterexample was found.* We repeat the process until there is no more counterexample or the feature quantifier becomes unsatisfiable. In the latter case, it means that

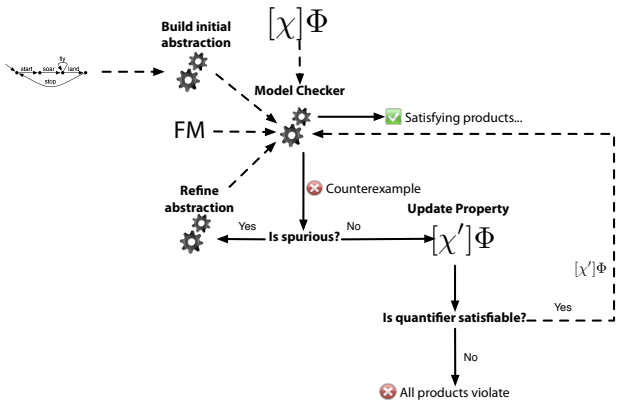


Figure 6: Overview of RWFO

all the products violate the formula.

The following theorem establishes the termination and the correctness of our CEGAR strategies.

Theorem 9 *Let h be an existential abstraction function, and h' be a refinement procedure. For any FTS, if Equations 2 hold then the CEGAR strategies terminate, are sound and are complete.*

PROOF. (Termination) *At the end of a verification, the CEGAR procedures either terminate, update the feature quantifier χ into χ', or trigger a refinement. Given that $[\![\chi']\!] \subseteq [\![\chi]\!]$ the number of updates is finite. By Equations 2, the number of refinement is also finite.*

(Correctness) *Let p be a valid product. After a verification, three cases may occur:*

1. p is reported as satisfying the formula. By Definitions 6 and 7, p satisfies the formula in the concrete FTS as well.

2. p is associated to a real counterexample, and thus violates the formula in the concrete FTS as well.

3. p is associated to spurious counterexamples only. In this case, p will be checked again after the refinement. Furthermore, the definition of χ' guarantees that only products that are known to satisfy or violate the formula are ignored in upcoming verifications. □

The strategies are agnostic to the actual abstraction and refinement functions being used, as long as these satisfy Definition 7 and Equations 2. In the following section, we show how to actually build different types of abstraction from a concrete model.

4. BUILDING FTS ABSTRACTIONS

Existing abstraction techniques generally do not work directly on TS. Instead, they rely on higher-level representations that include an explicit notion of variable [11]. These are commonly named *program graphs* [5]. In the context of SPLs, the products will be represented by a set of programs that share commonalities. In order to specify all these programs in a single compact model, one can borrow the concept of feature expression introduced in FTS and apply it to program graphs [14].

Definition 10 *(recalled from [14]) Let $V = \{v_1, \ldots, v_n\}$ be a set of variables, $\tau : V \to Types$ a type function, $Pred(V)$ the set of predicates over V, $Asgn(V)$ the set of assignments over V, and $Eval(v_1 \ldots, v_n) = \tau(v_1) \times \cdots \times \tau(v_n)$ the set of variable valuations. A Featured Program Graph (FPG) is a tuple $(Loc, V, \tau, Act, Loc_0, init, trans, d, \gamma)$ where Loc is a set of locations; $Act = Pred(V) \cup Asgn(V)$ is the set of actions; Loc_0 is the initial location; $Init \subseteq Eval(V)$ are the set of possible initial valuations; $trans \subseteq Loc \times Act \times Loc$ is the transition relation; d is an FM over features F; and $\gamma : trans \to \mathbb{B}(F)$ associates each transition with a feature expression.*

One can define the semantics of an FPG in terms of FTS (see [14] for details). Intuitively, given an FPG the state space of the underlying FTS is $Loc \times Eval(V)$, that is, an FTS state is defined as a location and a variable valuation. The set of initial states is $Loc_0 \times Init$. Transitions are determined according to the variable values of source state, as well as to the predicates and assignments that label the transitions of the current FPG location. Formally, the transition relation is the smallest relation satisfying:

$$\frac{l \xrightarrow{p} l' \wedge v \models p}{(l,v) \xrightarrow{p} (l',v)}$$

$$\frac{l \xrightarrow{x:=expr} l' \wedge v' = [x := expr]v}{(l,v) \xrightarrow{x:=expr} (l',v')}$$

where $[x := expr]v$ is v after assigning to x the value given by $expr$, and p is a predicate over the variables. Given a formula to check, the set AP of atomic propositions is the set of predicates over V that occur in the formula. Feature expressions are directly obtained from γ. In what follows, we interchangeably use γ to denote the transition labelling function of both an FPG and its underlying FTS. We now present two families of methods to build and refine abstractions of FPG, and discuss their implications on the CEGAR strategies presented above.

4.1 Feature Abstraction

The first abstraction type we propose consists in abstracting from the variability between the products. Concretely, we replace each feature expression $\gamma(t)$ labelling a transition t by another feature expression $\gamma_h(t)$ such that $[\![\gamma(t)]\!] \subseteq [\![\gamma_h(t)]\!]$. The set of states and transitions are, however, left untouched. The existential abstraction is thus defined as $h : S \to S : h(s) = s$. This transformation does not remove behaviours from the products, by Definition 7 and Theorem 8.

For the initial abstraction, we propose to completely abstract all the feature expressions of the FPG, that is, each of them is replaced by $\top$. This comes down to associating to every product of the underlying FTS (S, Act, $trans$, I, AP, L, d, γ) the semantics of the TS (S, Act, $trans$, I, AP, L). An undeniable advantage of this abstraction is that single-system model checking algorithms can perform the verification. If the initial abstraction satisfies a given formula then all the products satisfy it. Otherwise, the model checker returns a counterexample. Given that the F-abstraction did not modify the state space, the counterexample is necessarily an existing execution in the aforementioned TS. It may happen that no product can execute it (e.g., due to incompatible feature expressions), though; indeed, the TS semantics subsumes the union of the semantics of the projection of the FTS onto all valid products [18]. For instance, an execution that triggers transitions t and t', with $\gamma(t) \Leftrightarrow \neg\gamma(t')$ would be executed in the abstract FTS but not in the concrete FTS. Even if at least one product can execute the

counterexample, it does not mean that all the products can. More generally, if b_h (resp. b) is the feature expression characterizing the set of products that can execute the counterexample in FTS_h (resp. FTS), then the counterexample is spurious for products in $(\llbracket b \rrbracket \setminus \llbracket b_h \rrbracket) \cap \chi$.

If the counterexample is spurious for at least one product, the abstraction must be refined. Here, the refinement consists in replacing the abstract feature expression of each FPG transition that is executed during the counterexample by its concrete feature expression. This implies that for each FTS transition t, the feature expression labelling t, noted $\gamma_{h'}(t)$, is such that $\llbracket \gamma_{h'}(t) \rrbracket \subseteq \llbracket \gamma_h(t) \rrbracket$. Hence, the resulting abstraction function h' guarantees that $\llbracket FTS_{h'} \preccurlyeq_{FTS} FTS_h \rrbracket = \llbracket d \rrbracket_{FM}$ and $\llbracket FTS \preccurlyeq_{FTS} FTS_{h'} \rrbracket = \llbracket d \rrbracket_{FM}$ where h is the initial F-abstraction. Since we updated the feature expression of at least one transition, it follows that $\llbracket FTS_h \preccurlyeq_{FTS} FTS_{h'} \rrbracket \subset \llbracket d \rrbracket_{FM}$. Equations 2 are thus satisfied, which guarantees the termination and the correctness of the aforementioned CEGAR procedures.

4.2 State Abstraction

Unlike the first one, the second abstraction type relies on transposing the principles of classical abstraction functions to SPL CEGAR. More precisely, we consider *predicate abstraction*, one of the most applied methods for program abstraction [27, 11, 13]. Let $Pred = \{p_1, \ldots, p_n\} \subseteq Pred(V)$ be the set of predicates that occur in the FDG or in the formula to check. Let $h : (Loc \times Eval(V)) \to (Loc \times 2^{Pred})$ be an existential abstraction function such that $h(l, v) = (l, \{p_i \in Pred \bullet v \models p_i\})$. Intuitively, FDG locations are preserved in abstract states but a variable valuation is abstracted by the exact subset of predicates it satisfies. Since the predicates of the formula to check are included in $Pred$ and thus contribute to the definition of h, we have $h(s) = h(s') \Rightarrow L(s) = L(s')$ and $L(h(s)) = L(s)$. The abstract transition relation $trans_h$ is defined as the smallest relation satisfying:

$$\frac{l \xrightarrow{p} l' \wedge p \in P}{(l,P) \xrightarrow{p} (l',P)}$$

$$\frac{l \xrightarrow{x:=expr} l' \wedge ([P] \wedge wp(x := expr, [P']) \not\models \bot)}{(l,P) \xrightarrow{x:=expr} (l',P')}$$

where $[P] = \bigwedge_{p \in P} p \wedge \bigwedge_{q \in Pred \setminus P} \neg q$ and $wp(x := expr, \phi)$ is ϕ with each occurrence of x replaced by $expr$. Intuitively, P' is the set of predicates that are satisfiable after assigning $expr$ to x given that every predicate in P was satisfiable before the assignment and that every predicate not in P was not satisfiable. In this abstraction, we do not change the feature expressions, that is, $\gamma_h(t) = \gamma(t)$ for any transition t. Therefore we have $\llbracket FTS \preccurlyeq_{FTS} FTS_h \rrbracket = \llbracket d \rrbracket_{FM}$.

Technically, the definition of state abstraction is very similar to state-of-the-art predicate abstractions. The difference rather lies in the detection and the analysis of counterexamples, which now depends on variability. Indeed, the notion of spuriousness is more subtle in this case. First, a spurious counterexample may not correspond to an existing sequence of transitions in the concrete FTS. Second, it can be an existing execution but only for a subset of the products. We thus propose a method to detect spurious counterexamples that can be represented as finite sequences. This method is inspired by the SplitPATH algorithm used in single-system CEGAR [11]. It consists in identifying a set of states whose merging has yielded a spurious counterexample, and then splitting this set so as to make the counterexample disappear. We can also extend our technique with support for any kind of counterexample using unwinding techniques along the lines of Clarke *et al.* [11]. The key idea of our method is to remember, while replaying the counterexample, for which products we can execute each step of the replay. Formally, let $\sigma = h(s_0)\alpha_0 \ldots h(s_m)$ be a counterexample and σ_b the feature expression characterizing the products able to execute it. Let $S_0 = \{(i, \chi) \mid i \in I\}$ and

$$S_i = \{(s,b) \in h^{-1}(h(s_i)) \times \mathbb{B}(F) \mid (s',b') \in S_{i-1} \wedge \big(b = (b' \wedge \bigvee_{s' \xrightarrow{\alpha_{i-1}} s} \gamma(s', \alpha_{i-1}, s))\big)\}.$$

Intuitively, if $(s,b) \in S_i$ then s can be reached by products in $\llbracket b \rrbracket$ by replaying the first i steps of the counterexample. If for a product p there does not exist $(s,b) \in S_i$ with $p \in \llbracket b \rrbracket$ then p cannot execute the counterexample. Thus σ is spurious for products $\{p \in \llbracket \sigma_b \rrbracket \mid \nexists (s,b) \in S_m \bullet p \in \llbracket b \rrbracket\}$, or equivalently $\llbracket \sigma_b \wedge \neg(\bigvee_{(s,b) \in S_m} b) \rrbracket$. We also name these products spurious. The correctness of this detection procedure is derived from the proof of SplitPATH [11] and the definition of projection [16].

To refine the abstraction, we first have to identify the execution steps during which products are discovered not to be able to execute the counterexample. Let $\sigma_{b'}(i)$ present the products that can execute the counterexample in the concrete FTS up to step i, with $\sigma_{b'}(0) = \chi$. Then for any $i > 0$ the feature expression $\sigma_b \wedge \neg\sigma_{b'}(i)$ characterizes the products that are spurious at step i, and $\sigma_b \wedge \sigma_{b'}(i) \wedge \neg\sigma_{b'}(i+1)$ represents the products that became spurious exactly at step i. If the latter feature expression is satisfiable, a refinement should be performed at step i.

In predicate abstraction, a refinement consists in adding a new predicate in the construction of the abstraction function. An abstract state will therefore contain more information and the abstraction function will be able to produce finer abstractions. The predicate to add can be determined by means of Craig interpolation [24]. Given two formulae ϕ and ψ with $\phi \wedge \psi \models \bot$, a Craig interpolant is a formula ϕ' written using the intersection of the vocabularies of ϕ, ψ such that $\phi \Rightarrow \phi'$ and $\phi' \wedge \psi \models \bot$. Loosely speaking, ϕ' can be seen as an explanation of why $\phi \wedge \psi$ is not satisfiable. The occurrence of a spurious product means that the abstraction has created a transition between a state $s_i \in S_i$ and a state $s_{i+1} \in S_{i+1}$ that does not actually exist. To eliminate this spurious counterexample, the new abstraction has to make the distinction between s_i and the states in S_i that actually have a transition to s_{i+1}. We thus compute a Craig interpolant between $\bigwedge_{v \in V} v = eval(s_i, v)$ and $\bigvee_{(s_j \bullet s_j \xrightarrow{\alpha} s_{i+1})} \bigwedge_{v \in V} v = eval(s_j, v)$. This yields the new predicate to add as a refinement to the abstraction. Once the refined abstraction is obtained, we perform a new verification limited to the spurious products.

Let h' be the refined abstraction yielded by that method. Then $h'(s) = h'(s') \Rightarrow h(s) = h(s')$ and the abstract transition relation in $FTS_{h'}$ is finer than in FTS_h. Furthermore, the refinement method removes from at least one product the ability to execute the ith step of the spurious counterexample. Hence Equations 2 hold.

4.3 Mixed Abstraction

Given that the above two abstraction methods do not modify the same constructs, we can combine them to form coarser abstractions. Thereby, we obtain an initial existential abstraction function h as defined in Subsection 4.2 together with an abstract labelling function γ_h as presented in Subsection 4.1. Although the construction of the abstraction is straightforward, the detection of spurious counterexamples and the refinement process become more complex. Indeed, spurious products may originate from different factors, *viz.* the abstraction of feature expressions, the predicate abstraction, or their combination. This raises the questions of how to associate spuriousness cases with their appropriate origin(s), and how to guide the refinement process in case of multiple origins.

As before, we will compute sets of states S_i that give the states that can be reached upon the execution of the first i steps of the counterexample, together with the products able to reach them. The actual definition of S_i is as previously. However, since we have to take into account that feature expressions are abstracted as well, the detection algorithm should determine the reason why a product became spurious and refine either the labelling function or the state space abstraction.

Procedure IsSpurious formalizes our detection and refinement method. It consists in replaying the counterexample step by step, and check during each step whether we discover spurious products (Lines 5–19). At each iteration, σ_b represents the products that violate the property in fts_h and have not been discovered to be spurious, whereas $\sigma_{b'}$ represents the products that can execute the counterexample in fts up to the current step. For a given step i, we first check whether there are new spurious products due to the abstraction of feature expression (Lines 6–12). To achieve that, we compute the feature expression that would label the abstract transition should the feature abstraction not have been applied, add it to $\sigma_{b'}$ and compare the result with σ_b. The condition at line 8 means that any product in $[\![\sigma_b]\!] \setminus [\![\sigma_{b'}]\!]$ is a spurious product. If there is at least one such product, a refinement of the feature abstraction function is needed. Accordingly, we modify the feature expression labelling the abstract transition. Next, we check whether predicate abstraction yielded additional spurious products at step i by using the method presented in Subsection 4.2 (Lines 13–18). If it did (Lines 15–18), we compute and register the interpolant that will act as a refinement of the current predicate abstraction. The algorithm ends up returning either true if no refinement was needed, or a new F-abstraction together with the set of really violating products (Lines 21–22). This F-abstraction is built according to the refinement procedures of the previous two abstractions.

Input: fts and fts_h such that $[\![fts \preccurlyeq_{FTS} fts_h]\!] = [\![d]\!]_{FM}$, a quantifier χ, $\sigma = \widehat{s_0}\alpha_0 \dots \widehat{s_m}$ and $\sigma_b \in \mathbb{B}(F)$ such that $\sigma \in Prefix([\![(fts_h)_{|p}]\!]_{TS})$ for all $p \in [\![\sigma_b]\!]$.
Output: *True* if $\sigma \in Prefix([\![(fts_h)_{|p}]\!]_{TS})$ for all $p \in [\![\chi \wedge \sigma_b]\!]$, $(h', \sigma_{b'})$ such that Equations 2 hold and $\sigma \in Prefix([\![(fts_h)_{|p}]\!]_{TS})$ for all $p \in [\![\chi \wedge \sigma_b]\!]$ otherwise.

```
1  σ_b' ← χ ∧ d;
2  refined ← ⊥;
3  Pred' ← Pred;
4  γ_h' ← γ_h;
5  for i = 1 to m do
6  |  γ_h'(ŝ_{i-1}^, α_{i-1}, ŝ_i) ←
   |  ⋁_{s_{i-1}∈h^-1(ŝ_{i-1}^), s_i∈h^-1(ŝ_i)} γ(s_{i-1}, α_{i-1}, s_i);
7  |  σ_b' ← σ_b' ∧ γ_h'(ŝ_{i-1}^, α_{i-1}, ŝ_i);
8  |  if σ_b ⇏ σ_b' then
9  |  |  γ_h(ŝ_{i-1}^, α_{i-1}, ŝ_i) ← γ_h'(ŝ_{i-1}^, α_{i-1}, ŝ_i);
10 |  |  σ_b ← σ_b ∧ σ_b';
11 |  |  refined ← ⊤;
12 |  end
13 |  σ_b' ← σ_b' ∧ ⋁_{(s,b)∈S_i} b;
14 |  if σ_b ⇏ σ_b' then
15 |  |  Pred' ← Pred' ∪ {Craig(ŝ_{i-1}^ ∧ α_{i-1}, ŝ_i)};
16 |  |  σ_b ← σ_b ∧ σ_b';
17 |  |  refined ← ⊤;
18 |  end
19 end
20 if refined then
21 |  (h', γ_h') ← abstract(fts, h, Pred', γ_h');
22 |  return (h', γ_h');
23 else
24 |  return ⊤;
25 end
```

Procedure IsSpurious($fts, fts_h, \chi, \sigma, \sigma_b$)

5. IMPLEMENTATION AND EVALUATION

In order to compare and evaluate the potential benefits of our CEGAR procedures and our F-abstractions, we implemented them on top of our previous model checking algorithms [18, 16]. In this section, we describe our implementation, present the results of experiments we carried out, and attempt to infer general cases where it is particularly rewarding to apply CEGAR instead of standard FTS algorithms.

ProVeLines [23] is a product line of model checkers for SPLs, which constitutes the realization of a four-year research effort. It allows one to verify the behaviour of an SPL modelled as (an extension of) FTS against properties expressed in different logics. The variants of ProVeLines are semi-symbolic model checkers: they encode the SPL products symbolically as feature expressions, but explore the state space explicitly. Among the input languages accepted by ProVeLines one finds fPromela [14], a feature-aware extension of Promela [29]. fPromela provides constructs to declare variables and statements, as well as to label the latter with feature expressions. It can thus be regarded as a concrete syntax for featured program graphs, and is consequently appropriate as an input to our CEGAR procedures.

On the basis of our implementation, we carried out experiments to evaluate in which cases and to what extent our CEGAR-based verification methods are more efficient than a plain application of the FTS algorithms presented in our previous work [18, 16]. The benefits of CEGAR originate from two factors. First, the existential abstraction reduces the size of the state-space, and thus the number of states to explore before discovering a counterexample. Second, the abstraction of feature expressions increases the commonality between the behaviour of the products and thereby augments the potency of SPL-specific optimizations such as late splitting (see Section 2). However, the approximation due to abstraction and the consequent needs for refinement may

cancel these benefits, or even worsen the verification time. Indeed, after applying a refinement, a completely new verification must be performed on the refined model. After several refinements, the number of states explored by all the verifications altogether can exceed that of the standard algorithms. The performance of the CEGAR procedures thus depends on several factors including the topology of the model, the chosen F-abstractions, and the property.

As an attempt to estimate the impact of these factors, we computed the time needed by both CEGAR strategies combined with different F-abstractions and implemented on top of the FTS algorithms to verify three systems against a set of properties. We systematically compare the results with the time needed by the FTS algorithms without CEGAR to verify the same systems against the same properties. The first system is a minepump system SPL which has to draw water from a mine while there is no methane within it (see [31, 18, 16] for more information). This SPL consists of 11 features and 128 valid products. To explore the FTS modelling its behaviour, visiting 250,561 states is required. The second SPL is an elevator model inspired from Plath and Ryan [35, 17]. It is composed of eight features, which can be combined into 256 different products, and its FTS has 58,945,690 states to explore. The third and last SPL is a case study inspired by the CCSDS File Delivery Protocol (CFDP) [20], which has been reengineered as a product line [8]. The FTS modelling the protocol consists of 1,801,581 states to explore and 56 products.

We focus first on the minepump SPL. We checked the corresponding FTS against 20 properties, including deadlock freedom, safety and liveness properties. Due to lack of space, we do not display detailed results for the other properties and case studies. However, all the results together with our case studies and implementation are available on our website[1]. All benchmarks were run on a MacBook Pro with a 2,8 GHz Intel Core i7 processor and 8 GB of DDR3 RAM running Mac OS 10.7. We coded an automated script to execute them. To avoid random variations, we repeated each experiment five times and computed the average. We discuss the results in terms of *speedup*, *i.e.* the verification time using a given abstraction divided by the verification time of the standard FTS algorithm. The verification time includes the time to parse the fPromela model, to build the initial FTS, and to run the verification procedure.

The results are presented in Figure 7, which shows the speedup of each CEGAR strategy and abstraction with respect to the standard algorithm. When feature abstraction is applied together with the FABR strategy, the model checker performs significantly better than the standard FTS algorithm (i.e. with a speedup comprised between 1.38 and 11.07) in five cases out of 20; it has almost no effect (speedup between 0.9 and 1.1) in eight cases; and it performs slightly worse (speedup between 0.77 and 0.86) in the last seven cases. The results are very similar when the RWFO strategy is followed instead. In both cases, after an in-depth analysis of the returned counterexamples we noticed that the cases where the abstraction does not improve the performance occur when every feature has to be known for the counterexamples to be triggered. This means that after several refinements the model checker ends up verifying the original concrete FTS. The small loss of performance is due to the verifications performed before the last refinement. An impressive speedup of more than nine is achieved when the property is satisfied by all products (three cases); when only a few features are needed to find the counterexamples (three cases), the abstraction still brings nice performance gains (speedup between 1.38 and 1.91). A change of strategy brings but small variations in the result; this factor thus does not seem to impact the overall efficiency of feature abstraction.

The topology of the system, however, substantially affects the efficiency of this form of abstraction. In the elevator system, each feature leads to behavioural variations that occur at the beginning of the execution of the system; yet all the products have a lot of common behaviour. This implies that late splitting occurs early in the verification and that the algorithm has to explore more than once a large part of the state space. By abstracting feature expressions, we can drastically reduce these re-explorations. Concretely, the abstraction performed significantly better when verifying six properties out of 19, achieving impressive speedups (82.12 and 93.38) in the two cases where the property is satisfied by all products. In these cases, the verification time was reduced from 293.17 and 984.32 to 3.57 and 10.54 seconds, respectively. As for the remaining 13 properties, almost no variation with respect to the standard algorithm were reported. On the contrary, the CFDP case study gives a lot of trouble to feature abstraction. The FTS modelling the protocol is uncommon, as a very large part of the state space is available to only one product. Abstracting feature expression results in the addition of such a large behaviour to all the other products, which will be explored for each of them. For one property out of six, this inconvenience did not affect the results. However, it led to a significant decrease in efficiency in the five other cases, with a speedup between 0.14 and 0.46.

State abstraction appears more invasive than feature abstraction. When combined with the FABR strategy, it had no significant effect on the performance for only four properties of the minepump SPL; it improved it nine times, and worsened it seven times. In the latter cases, the loss in efficiency is substantial (speedup between 0.12 and 0.79), whereas the improvements yield speedups between 1.17 and 2.50. Our observations tend to indicate that RWFO is more appropriate for state abstraction. Indeed, the performance of the abstraction is always increased when combined with this strategy. With respect to the standard algorithm, the verification is improved for 15 properties (speedup between 1.35 and 2.57), worsened for four properties (speedup between 0.27 and 0.68), and almost unchanged for one property. These conclusions are confirmed by the elevator and CFDP case studies, from which we gathered similar results.

It is noteworthy that the above two abstractions have both a negative impact on verification time for only one property. Moreover, it happened several times that one form of abstraction was very inefficient but not the other. This implies that (1) there almost always exists a good choice regarding the abstraction to apply and (2) their combined effects cannot be inferred from their individual performance results. Therefore we carried out additional experiments on mixed abstraction. It turned out that the results follow the same trend as in the case of state abstraction, although mixed abstraction is slightly less efficient on average. State abstraction thus seems to cancel the effect of feature ab-

[1] `http://info.fundp.ac.be/fts`

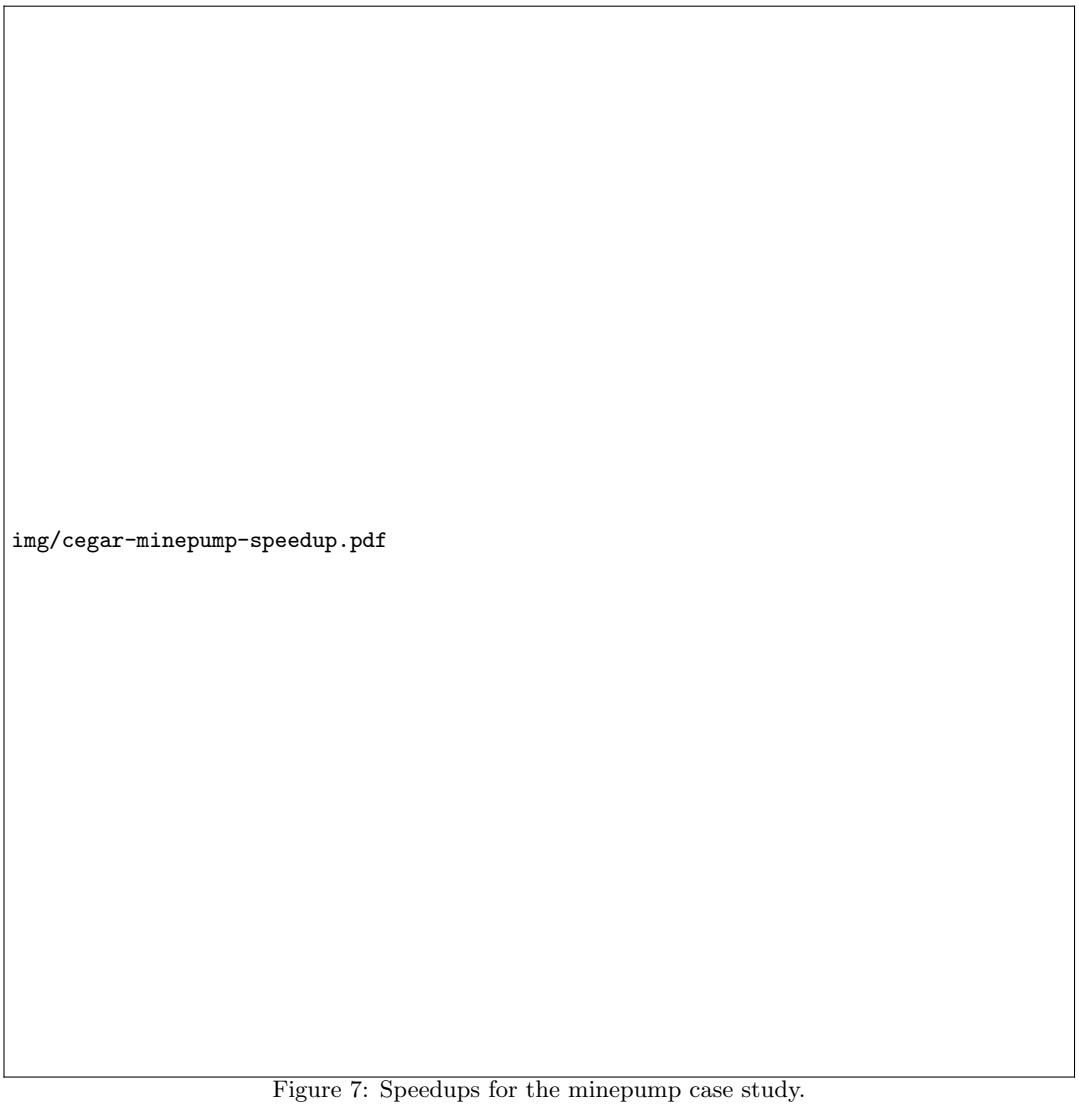


Figure 7: Speedups for the minepump case study.

straction, be it negative or positive. This corroborates our previous observation that state abstraction has more impact on performance. An in-depth look at the execution times revealed that the computation of successor states in state abstraction is far more costly than the re-exploration of already computed states. Since feature abstraction improves the latter but not the former, its effects are not apparent in mixed abstraction. The other two case studies tend to confirm these conclusions. In particular, the negative effects of feature abstraction on the CFDP model are offset by the benefits of state abstraction, although mixed abstraction remains seemingly less efficient than state abstraction alone in this case.

Conclusions. All these experiments allow us to draw conclusions regarding which abstraction to use. State abstraction brings substantial benefits in most cases and should be the preferred form of abstraction. Still, feature abstraction can achieve dramatic reductions in verification time in particular situations, notably when the property to check is (expected to be) satisfied by all products (see, *e.g.*, property #8). The performance of our mixed abstraction is of the same order as state abstraction, but is less efficient on average. Yet, mixed abstractions offer the highest level of customization; it could thus be possible to define other mixed abstractions that are more efficient than the one we used. We leave that for future work.

6. RELATED WORK

This work is at the intersection of CEGAR and product-line model checking. Clarke *et al.* [11] were the first to introduce CEGAR. They presented the principles we recapitulated in Section 2, designed the SplitPATH algorithm used to detect spurious counterexamples, and proposed a refinement algorithm. Several model checkers make use of predicate abstraction to speed up the verification. At the last TACAS software verification competition [6], UFO [1], LLBMC [25], CPAChecker [7], and ESBMC [34] were nominated the fastest model checkers for product lines. Unlike us, these tools do not treat features as first-class citizens. This means that either they rely on an enumerative approach or their analysis procedure consists in verifying a model that includes the behaviour of all products (i.e. based on a product simulator [2] or on configuration lifting [37]). In the first case, abstraction can be applied to the models of the individual products but the commonality between these is not exploited. The second case offers interesting perspectives regarding an efficient application of abstraction on all products. However, this approach can determine whether or not *all* products satisfy the property, while we want to identify *which* products are error-prone. Outside the scope of model checking, Liebig *et al.* [32] designed a type checking and data-flow analysis procedure that abstracts from the validity of products. This is also a form of feature abstraction, although ours goes further by completely abstracting features.

Regarding SPL model checking, there exist other methods that are not based on FTS. Fischbein *et al.* [26] proposed to use Modal Transition Systems (MTS) to model the behaviour of SPLs. MTS are TS where transitions are either mandatory (*i.e.* executable by all valid products) or optional (*i.e.* executable by only a subset of valid products). A fundamental difference between MTS and FTS is that the former cannot link a given execution to the exact set of products able to produce this execution. It is thereby impossible to identify the exact set of products that violate a given property. To overcome this limitation, Asirelli *et al.* [4] equipped MTS with a modal logic which permits to restrain the execution of actions to specific combination of features. Grumberg *et al.* studied an abstraction-refinement model checking procedure for modal μ-calculus, whose principles could be reused to design CEGAR methods for standard MTS. However, such a method would not benefit from the modal logic of Asirelli *et al.* and would thus be inappropriate to address the SPL model checking problem.

Apel *et al.* [2] developed SPLVerifier, a tool chain for product-line model checking. Features are specified in separate modules written in C or Java. Like [1, 25, 7, 34], the advantage of their approach over FTS is that they can verify actual code. However, the properties they can check do not extend to logics as expressive as LTL. In [3], they showed that approaches relying on symbolic interpretation of features are more efficient than sample-based approaches, thereby corroborating our previous results [14, 16].

Gruler *et al.* [28] showed that multi-valued model checking can address the SPL model checking problem. Multi-valued TS generalise TS in that the transition relation is not binary and the atomic propositions are not Boolean [10, 9]. FTS can be regarded as a particular type of multi-valued TS. There currently exists no algorithm to efficiently verify LTL formulae on multi-valued models (see [16] for a thorough comparison). Given the close relation between FTS and multi-valued models, the principles presented in this paper can be applied to design CEGAR procedures for multi-valued model checking.

7. CONCLUSION

We presented SPL-specific abstraction methods that tackle the state explosion problem in SPL model checking. Our evaluation tends to show substantial performance gains in a majority of cases, from which we derived general rules as to when our heuristics should be applied. In the future, we plan to design an automated method to build appropriate mixed abstractions given the topology of the model to check and a history of previous verifications. We will also combine this work with our fully symbolic SPL model checking algorithms and implement the resulting approach on top of our NuSMV extension [17]. This will allow us to assess whether or not our conclusions are still valid in that case.

8. REFERENCES

[1] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *CAV*, pages 672–678, 2012.

[2] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Feature-interaction detection using feature-aware verification. In *ASE'11*, pages 372–375. IEEE, 2011.

[3] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: case studies and experiments. In *ICSE'13*, pages 482–491, 2013.

[4] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. Formal description of variability in product families. In *SPLC'11*, pages 130–139. Springer-Verlag, 2011.

[5] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2007.

[6] D. Beyer. Second competition on software verification - (summary of sv-comp 2013). In *TACAS '13*, pages 594–609, 2013.

[7] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV '11*, pages 184–190, 2011.

[8] Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, and L. Demonceau. Tag and prune: A pragmatic approach to software product line implementation. In *ASE'10*, pages 333–336. ACM, 2010.

[9] G. Bruns and P. Godefroid. Model checking with multi-valued logics. In *ICALP '04*, pages 281–293, 2004.

[10] M. Chechik, B. Devereux, and A. Gurfinkel. Model-checking infinite state-space systems with fine-grained abstractions using spin. In *SPIN '01*, pages 16–36, 2001.

[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer Berlin / Heidelberg, 2000.

[12] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[13] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ansi-c programs using sat. *Form. Methods Syst. Des.*, 25(2-3):105–127, Sept. 2004.

[14] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking software product lines with SNIP. *STTT*, 14(5):589–612, 2012.

[15] A. Classen, M. Cordy, P. Heymans, P.-Y. Schobbens, and A. Legay. Snip: An efficient model checker for software product lines. Technical report, University of Namur (FUNDP), 2011.

[16] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. cois Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *Transactions on Software Engineering*, pages 1069–1089, 2013.

[17] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *ICSE'11*, pages 321–330. ACM, 2011.

[18] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE'10*, pages 335–344. ACM, 2010.

[19] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, August 2001.

[20] Consultative Committee for Space Data Systems (CCSDS). *CCSDS File Delivery Protocol (CFDP): Blue Book, Issue 4*. NASA, 2007.

[21] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Managing evolution in software product lines : A model-checking perspective. In *VaMoS'12*, pages 183–191. ACM, 2012.

[22] M. Cordy, A. Classen, G. Perrouin, P. Heymans, P.-Y. Schobbens, and A. Legay. Simulation-based abstractions for software product-line model checking. In *ICSE'12*, pages 672–682. IEEE, 2012.

[23] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Provelines: A product-line of verifiers for software product lines. In *SPLC'13*, pages 141–146. ACM, 2013.

[24] W. Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[25] S. Falke, F. Merz, and C. Sinz. The bounded model checker llbmc. In *ASE '13*, pages 706–709, 2013.

[26] D. Fischbein, S. Uchitel, and V. Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA'06, ISSTA 2006 workshop*, pages 39–48. ACM Press, 2006.

[27] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83, London, UK, UK, 1997. Springer-Verlag.

[28] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and model checking software product lines. In *FMOODS'08*, pages 113–131. Springer, 2008.

[29] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[30] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990.

[31] J. Kramer, J. Magee, M. Sloman, and A. Lister. Conic: an integrated approach to distributed computer control systems. *Computers and Digital Techniques, IEE Proceedings E*, 130(1):1–10, 1983.

[32] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *ESEC/SIGSOFT FSE '11*, pages 81–91, 2013.

[33] R. Milner. An algebraic definition of simulation between programs. Technical report, Stanford University, Stanford, CA, USA, 1971.

[34] J. Morse, L. Cordeiro, D. Nicole, and B. Fischer. Handling unbounded loops with esbmc 1.20 - (competition contribution). In *TACAS*, pages 619–622, 2013.

[35] M. Plath and M. Ryan. Feature integration using a

feature construct. *SCP*, 41(1):53–84, 2001.

[36] A. Pnueli. The temporal logic of programs. In *FOCS'77*, pages 46–57, 1977.

[37] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *ASE'08*, pages 347–350. IEEE CS, 2008.

[38] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *RE'06*, pages 139–148, 2006.